Bike.ai

Fall 2019, EECS 149 Final Project Report

Bernard Chen, Arjun Mishra, Michael Duong

Objective

Bike.ai is a modular system providing non-intrusive brake and turn lighting, “blind-spot” proximity detection, and an accompanying iOS application. The system is designed with non-intrusiveness in mind, so all components are mostly wireless and can be easily attached to any bicycle without extensive wiring and fits to any and all bicycles.

Demo video: https://youtu.be/Q-gUB0No8FQ

Inspired by Neal (GSI)’s idea of a brake detection system, our project includes additional features to improve rider safety and experience. The brake light LED strip will turn on when the system detects a substantial deceleration indicating a brake. The left and right turn signal indicators can also be activated via wireless BLE buttons and will automatically turn off when a turn is made or a timeout occurs. Additionally, our system implements proximity detection through two ultrasonic sensors, that activate proximity indicator LEDs on the handlebars when an object is detected within a configurable distance. We go further by emphasizing the non-intrusive nature of our system and by offering customizability through an iOS application. We designed our system to be non-intrusive so we opted to not use a hall-effect sensor, which would require the user to measure the bicycle’s wheels and require extensive wiring. The iOS application allows a user to wirelessly configure and customize a brake detection mode, proximity sensing distance, turn signal colors, and brake light color.

Our project serves as a good foundation to be developed into a fully non-intrusive bicycle lighting and safety system. Immediate improvements could be packaging all components into a single device. This could include building a custom PCB and using a more dedicated and smaller nRF microcontroller. Additional stretch goals/features could include building out a platform for rider data collection (where data is harvested from sensors on the device and aggregated on some platform like a web application for the user to see) and the inclusion of an ambient light sensor and a dedicated light for night-time riding. Any feature can be easily added and interfaced with through BLE, which our current system already supports and provides a framework for.

System Design Overview

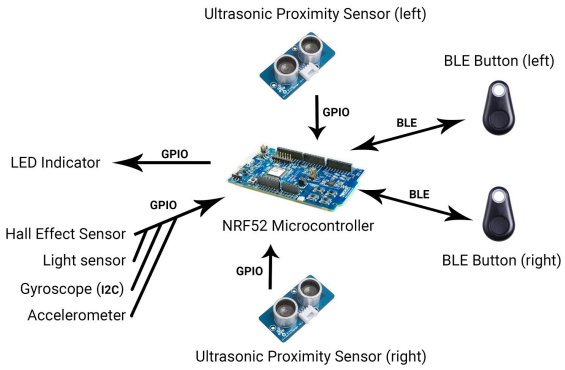


Figure 1: System architecture

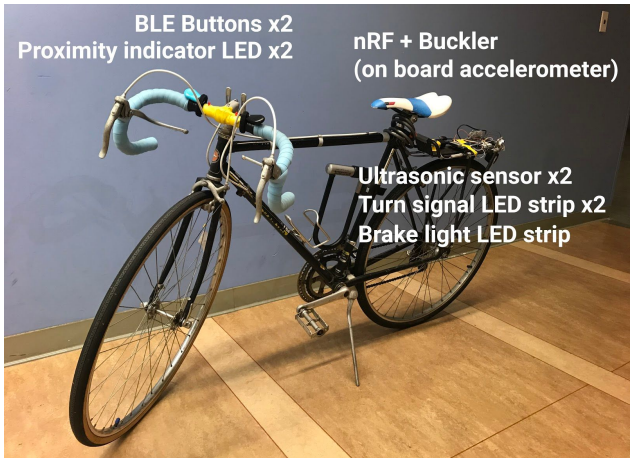


Figure 2: Complete system. See figures 4 to 7 for individual components and subsystems.

Overview and work breakdown

As seen in figure 1, each component, such as the ultrasonic proximity sensor and the BLE buttons, integrates with the microcontroller directly. The components are independent subsystems, so that allowed us to build our system in a modular way. Each team member was in charge of one major component: automatic brake detection was handled by Arjun, the proximity “blind spot” detection by Michael, and the BLE turn signals by Bernard. Additionally, the iOS app was developed by Arjun and the GATT client-side code for the iOS app on the nRF was written by Bernard. Integration was primarily headed by Michael.

With our level of modularity, each subsystem was built and tested individually. Upon integration, bugs encountered were integration specific, thus making the issues easier to diagnose and allowing for a smoother process. Figure 2 shows the fully integrated system.

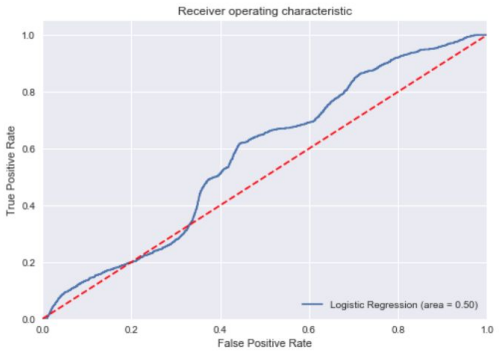
Automatic Brake Detection

For the automatic brake detection, we had three main ideas for approaches. The standard approach with streaming data is Online Outlier Detection. This involves calculating that a datapoint is an outlier in real time, which means that computations must be cheap. In order to facilitate this, we came up with two methods which we later validated on collected sample data, by logging onto an SD card. These methods utilize a sliding window of the last 30 observations, and continually find the 25% and 75% of this sliding window. A standard technique in statistics for outlier detection is to take the 75% percentile to 25% percentile namely the IQR and multiply it by some hyper parameter α which is then added to the 75% percentile to detect upper bound outliers and subtracted from the 25% percentile to detect lower bound outliers. This method shown in figure 10 proved effective for general use. The larger we could make the sliding window, the more accurate our predictions were. However, we encountered RAM overflow errors after integration with 30 doubles as our sliding window.

In order to find the 25th and 75th percentile in constant time, it is possible to implement a max heap and min heap and properly balance between these two. However in our case since we needed the median, 75th percentile, and the 25th percentile it was not possible to use such a run time optimization and instead we opted to quicksort after every new data point was added. Quick sorting an almost sorted array is near worst case runtime and insertion sort on smaller collections is also known to be faster. Therefore, we recommend that these optimizations be made for better performance in the future.

Finally we tested two IQR based methods: residual and non-residual. The main difference between these two is that the non-residual consists of a sliding window of actual accelerometer readings, whereas the residual based method consists of the $current\ observation/reading - previous\ median$, so that we have a sliding window of residuals. We then compare whether our current residual is an outlier based on IQR.

Additionally, we attempted applying machine learning techniques to see if we could serialize the weights of a logistic regression model. However, prior to the process of serialization, in our exploratory analysis on Jupyter, I found that such a logistic classifier was of limited accuracy on the minimal feature transformations I did on our test dataset, and therefore, it did not ultimately make sense to do it. In addition we felt that a moving average, median based model would better fit to all possible use cases, as different bikes may incur different forces which the logistic model may not generalize well too.

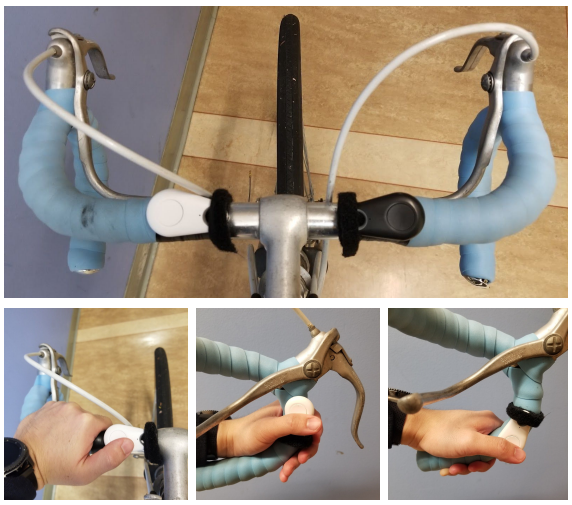



```
             precision    recall  f1-score   support

          0       0.95      1.00      0.98     36864
          1       0.00      0.00      0.00      1823

avg / total       0.91      0.95      0.93     38687

[[36864     0]
 [ 1823     0]]
Accuracy of logistic regression classifier on test set: 0.95
```

Figure 3: ROC (Receiver Operating Characteristic) curve of logistic regression techniques is shown to be no better than predicting at random.

BLE-activated Turn Signals

Our turn signals are activated completely wirelessly and built on BLE. Because we are using the nRF as a GAP central and the simple_ble library does not support using the nRF and Buckler as a central, we wrote custom drivers/firmware purely using the nRF SDK and SoftDevice API[3] to communicate with third-party iTag buttons.



Figures 4: Possible BLE turn-signal button placements. The flexibility is a result from a completely wireless design.

A framework was written to support BLE from our device, using specifically the scanning module and database discovery module from the nRF SDK. Once a connection was established with the desired peripherals, the nRF board acts as a GATT client, using the SoftDevice API to perform read and write operations.

The development of BLE buttons also had a few notable issues. We initially tried to use Flic buttons but after significant research and development, we found that Flic buttons had custom firmware that prevented us from using the pure BLE interface and only allowed connection through their designated integrations or the official Flic app. With the iTag buttons, we found that the CCCD (Client Characteristic Configuration Descriptor) was missing from the characteristics even though the notify property was enabled. Thus, we were unable to subscribe to button click notifications and instead used polling. Upon reading a 1 from the characteristic tied to the button click, the BLE event handler acknowledges that the button click was received by resetting the characteristic to a 0 in a write using sd_ble_gattc_write. We also explored using a second and third nRF and buckler as buttons so that we could control the firmware of the peripheral but ultimately found the solution too bulky.

LED Lighting

Significant time was spent working with the WS2812B LED strips. We wrote our own PWM driver from scratch to control the LED strips, which takes a very precise PWM data signal. Open source libraries that control the WS2812B like FastLED did not yet support nRF boards. The LED strips take 24 bits encoded in a PWM signal, where 40% high corresponds to a 0 and 60% high corresponds to a 1. This encodes three 8-bit RGB values, and the data must be held low for a precise time for the colors to "latch." [1] Configuring and debugging the LEDs ended up requiring a significant workload to drive them from the GPIO pins and configure/customize them with colors.

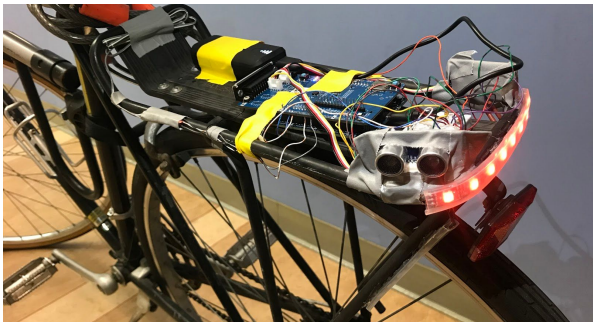


Figure 5: Brake light LED strip turned on when brake is detected. Also, the ultrasonic ranger for the proximity sensor can be seen above the brake LED.

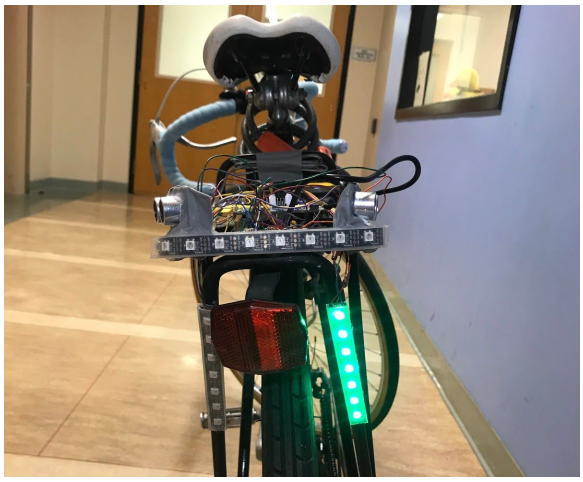


Figure 6: Three WS2812B LED strips indicating braking, left turn, and right turn, configurable to different colors.

Proximity "Blind Spot" Sensors

For our proximity sensors, we chose to use a Grove ultrasonic ranger to measure distance. The reason for this is that we did not just want to detect the presence of the object, but also how far away the object is, so that we have control of the range that an object is considered "close." If an object is detected to be in range, then that would turn on a proximity indication LED at the handlebars, as shown in figure 7.

Figure 7: Proximity LED indicators turned on. The sensors detect the wall on one side and a person on the other. See figure 5 for ultrasonic sensors.

In order to use the ultrasonic rangers to measure distance, we used the online documentation for the ranger[2] to figure out its proper usage. From that, we developed and implemented a state machine, as seen in figure 8, that carries out the process recommended by the documentation. We then check if the distance is within the configured threshold, and if it is, the LED is changed to the ON state.

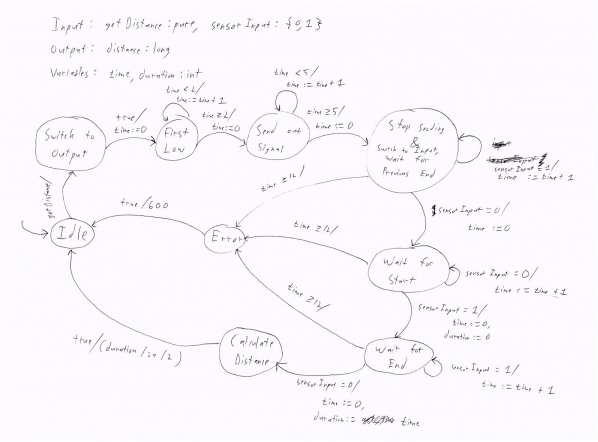


Figure 8: The state machine used to measure distance with the ultrasonic ranger. It switches the ranger to output, sends a signal, switches to input, reads the echo, calculates the distance, and outputs it.

iOS app: customizability through BLE

To enable customizability, we built an iOS application in Swift and developed a custom protocol to encode and send configurations to the nRF board through BLE in real-time. The nRF board again acts as a central. The application allows users to pick configurations for brake mode, proximity detection distance, and brake and turn light colors, and sends them over BLE to the nRF board. The iOS application on mobile acts as a peripheral and begins advertising to open connections when a user hits “Done.” The nRF board will seek a connection, connect, discover the correct service and read the configurations from the correct characteristic using the same firmware written for turn signals. Because we ran into busy errors as the SoftDevice wasn’t able to handle multiple packets for this, we developed an encoding to send all the settings in an 8-bit data packet. The leftmost two bits correspond to a turn light color, next two bits to a brake light color, next two bits to proximity distance and rightmost two bits to a brake detection mode.

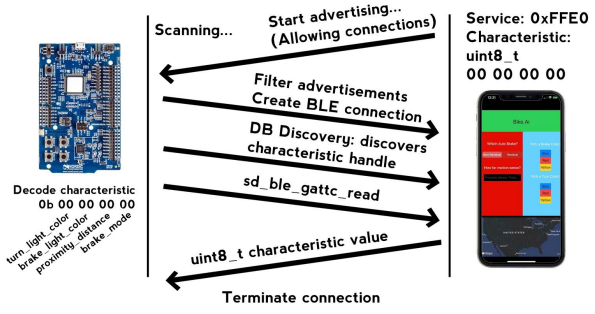


Figure 9: Our protocol for sending customizations through BLE from the iOS app.

Relation to Course Concepts

State machines

Hierarchical state machines were used heavily in the design of our project. Each subsystem is modeled with its own state machine with respective inputs from sensors and outputs to LEDs. Within these subsystems, the proximity sensor included a sub-state-machine to read the distance from the ultrasonic ranger, and the turn signals had a sub-state-machine to blink the LEDs. All of these subsystems were combined into one complete system.

Wireless communication

Wireless communication was implemented through BLE (Bluetooth Low Energy). Our buttons to activate turn signals were interfaced through BLE, allowing for a portable and non-intrusive solution. We also communicated with the iOS application to get user configurations through BLE. The simple_ble library provided in lab only allows the nRF to serve as a peripheral and does not support the nRF as a central, so we wrote and implemented custom firmware using the nRF SDK and SoftDevice API.

Interrupts

There was extensive usage of interrupts. We used timers to increment variables in our state machines, which were handled by interrupts. The BLE stack also heavily relies on interrupts, with event handlers for each BLE action. For example, we read from a GATT server through sd_ble_gattc_read, which should return successfully, but the actual value is retrieved in a BLE event handled by our ble_evt_handler. The same applies for connection, service/characteristic discovery, and other BLE interactions.

I/O (GPIO), Sensors, Actuators

All three of the subsystems receive inputs through sensors, like the accelerometer or ultrasonic ranger. Once the input is processed, the system produces output to actuate LEDs. All of our subsystems followed this input, process, output model, interacting with GPIO, which is a key concept for our project.

Memory Architecture, Timing, Multitasking

Our project was built entirely on the nRF52 microcontroller. We ran into issues with memory constraints because our brake detection algorithm stores a large window of samples. We also developed our device with timing in mind, such that everything could run in real-time and no subsystem would stall the entire system.

Implemented System Evaluation and Results

Our final system achieved all set goals and all subsystems were integrated without conflict. We acknowledge that our brake detection system is not completely robust, because we are using outlier detection and after a window of 30 full samples of zero acceleration, any slight movement is considered an outlier. Obvious braking and deceleration is detected, shown with non-residual and residual IQR outlier detection in figure 10. In our test data, brakes at around 4 seconds (80 samples) and 7 seconds (140 samples) were detected by the analysis, shown when the accelerometer value (middle line) intersects either the upper quartile or lower quartile lines.

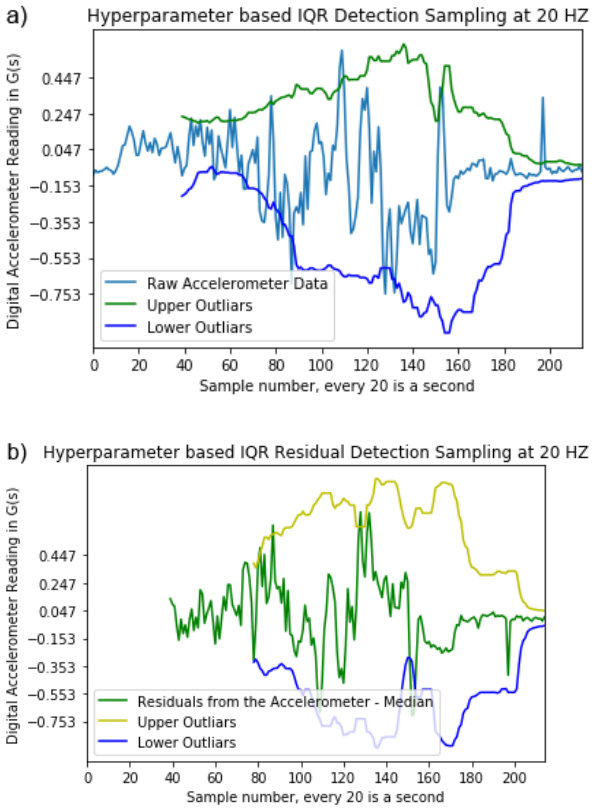


Figure 10: IQR outlier detection for a single test ride with braking at about 4 seconds (80 samples) and 7 seconds (140 samples) using (a) absolute value accelerometer readings and (b) residuals.

Our turn signals modeled with our state machine worked as expected, with BLE activating the signals and auto-turnoff when the button is pressed again, timeout occurs, or a turn is detected. Lastly, the ultrasonic ranger was able to withstand the noise on the streets and reliably detect when an object was in the rider's blind spot. The iOS app provides real-time customization for the rider through BLE.

Why features did and did not work

The BLE activated turn signals and ultrasonic rangers for proximity sensing work very reliably, especially because we were able to model them accurately and implement corresponding firmware. Integration also went smoothly because we designed our system with modularity in mind and each subsystem was debugged separately before integration. Automatic brake detection could be improved by keeping a larger window of samples. We were forced to reduce the number of samples for outlier detection to ~30 due to memory constraints on the nRF board (especially after integrating all our code) but could be improved by using floats to sacrifice accuracy and also allocating appropriate memory. This caused our method to be less reliable because fewer past accelerometer readings were used to detect outliers. Thresholding at some value to ignore outliers while the bike is not moving would also make the system more robust.

Conclusion

Overall, we built a system to detect when a bicyclist is braking, use BLE buttons to enable turn signals, monitor blind spots to warn riders of objects nearby, and communicate with an iOS app to customize the experience. Our brake detection occasionally has false positives, but it reliably detects intentional braking, and every other subsystem was reliable. Future improvements may include a more compact and packaged device and stretch goals like integrating an LED for night time, activated by a light sensor.

References

1. Burgess, Phillip. "Adafruit NeoPixel Überguide." *Adafruit*, https://learn.adafruit.com/adafruit-neopixel-uberguide/advanced-coding
2. "Grove - Ultrasonic Ranger." *Seeedstudio*, http://wiki.seeedstudio.com/Grove-Ultrasonic_Ranger/.
3. "nRF5 SDK v15.3.0." *Nordic Semiconductor*, https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.3.0%2Findex.html

Additional references can be found throughout our codebase. (i.e. Based on simple PWM code or from /examples/ble_central/ble_app_multilink_central from the nRF SDK)